Lecture #5: Higher-Order Functions

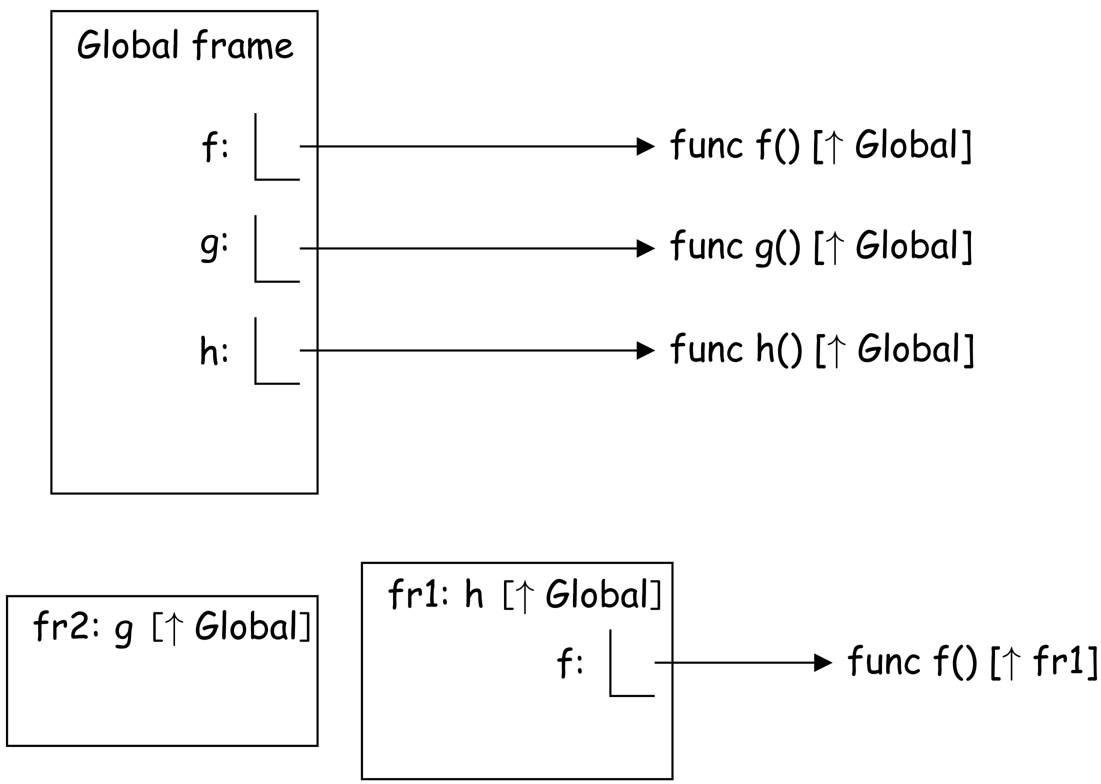
Do You Understand the Machinery? (I)

What is printed (0, 1, or error) and why?

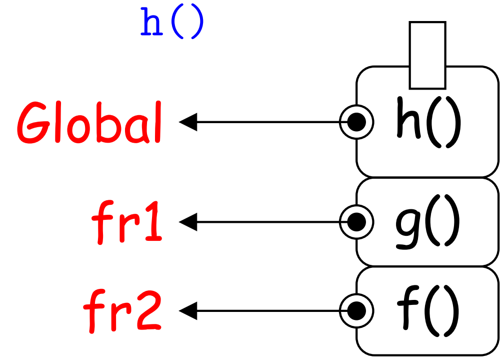
```
def f():
    return 0

def g():
    print(f())

def h():
    def f():
        return 1
    g()

h()
```

Answer (I)

The program prints `0`. At the point that `f` is called, we are in the situation shown below:




```
def f():
    return 0
def g():
    print(f())
def h():
    def f():
        return 1
    g()
h()
```



So we evaluate f in an environment (fr2) where it is bound to a function that returns 0.

Do You Understand the Machinery? (II)

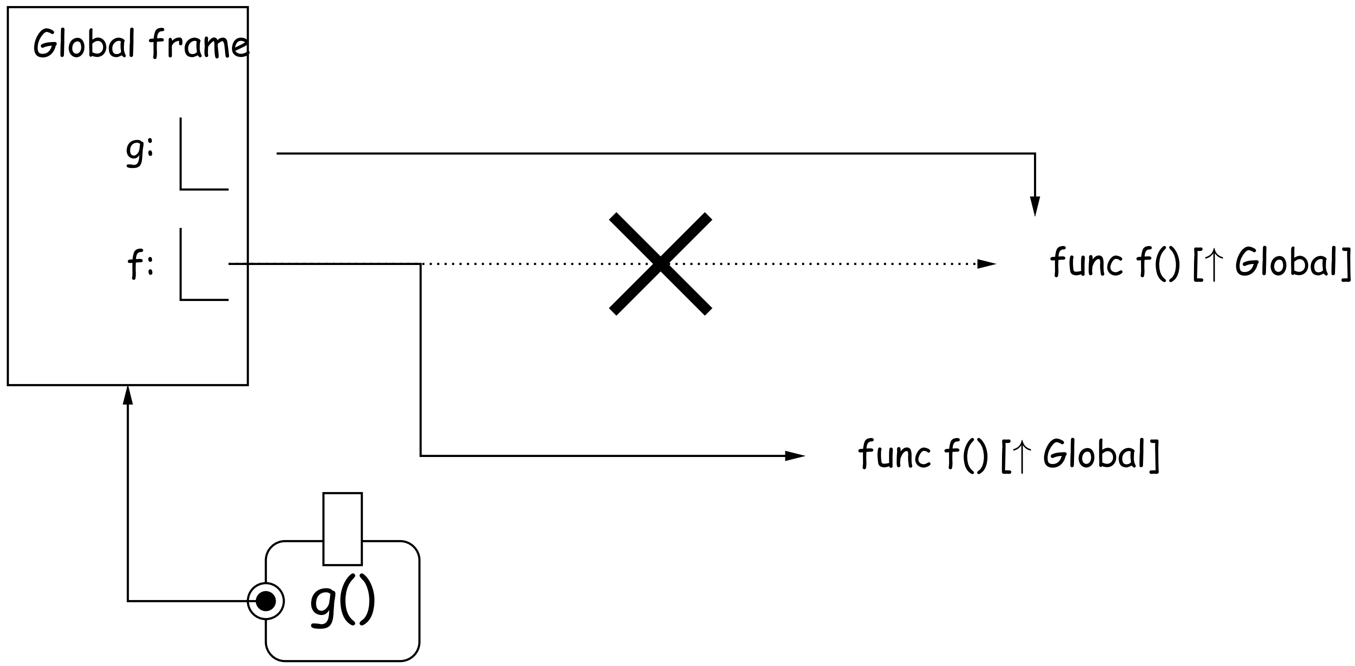
What is printed (0, 1, or error) and why?

```
def f():
    return 0

g = f

def f():
    return 1

print(g())
```

Answer (II)

The program prints 0 again:

```
def f():
    return 0

g = f

def f():
    return 1

print(g())
```



At the time we evaluate f in the assignment to g, it has the value indicated by the crossed-out dotted line, so that is the value g gets. The fact that we change `f`'s value later is irrelevant, just as x = 3; y = x; x = 4; print(y) prints 3 even though x changes: y doesn't remember where its value came from.

Do You Understand the Machinery? (III)

What is printed (0, 1, or error) and why?

```
def f():
    return 0

def g():
    print(f())

def f():
    return 1

g()
```

Answer (III)

This time, the program prints `1`. When g is executed, it evaluates the name 'f'. At the time that happens, f's value has been changed (by the third **def**), and that new value is therefore the one the program uses.

Do You Understand the Machinery? (IV)

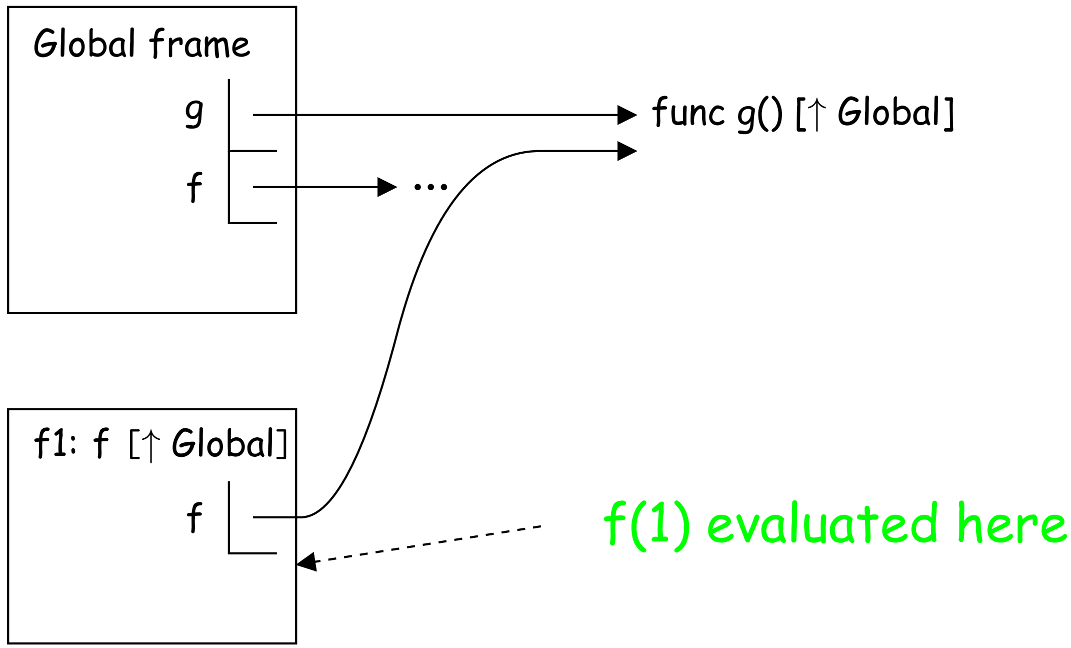
What is printed: (1, infinite loop, or error) and why?

```
def g(x):
    print(x)

def f(f):
    f(1)

f(g)
```

Answer (IV)

This prints 1. When we reach f(1) inside f, the call expression, and therefore the name f, evaluated in the environment E, where the value of f is the global function bound to g:

```
def g(x):
    print(x)

def f(f):
    f(1)

f(g)
```



Do You Understand the Machinery? (V)

What is printed: (0, 1, or error) and why?

```
def f():
    return 0

def g():
    return f()

def h(k):
    def f():
        return 1
    p = k
    return p()

print(h(g))
```

Answer (V)

This prints 0. Function values are attached to current environments when they are first created (by lambda or def). Assignments (such as to p) don't themselves create new values, but only copy old ones, so that when p is evaluated, it is equal to k, which is equal to g, which is attached to the global environment.

Observation: Environments Reflect Nesting

- From what we've seen so far:

  Linking of environment frames $\Longleftrightarrow$ *Nesting of definitions.*

- For example, given

```
def f(x):
    def g(x):
        def h(x):
            print(x)
        ...
    ...
```

  The structure of the program tells you that the environment in which *print(x)* is evaluated will always be a chain of 4 frames:

  - A local frame for *h* linked to ...
  - A local frame for *g* linked to ...
  - A local frame for *f* linked to ...
  - The global frame.

- However, when there are multiple local frames for a particular function lying around, environment diagrams can help sort them out.

Do You Understand the Machinery? (VI)

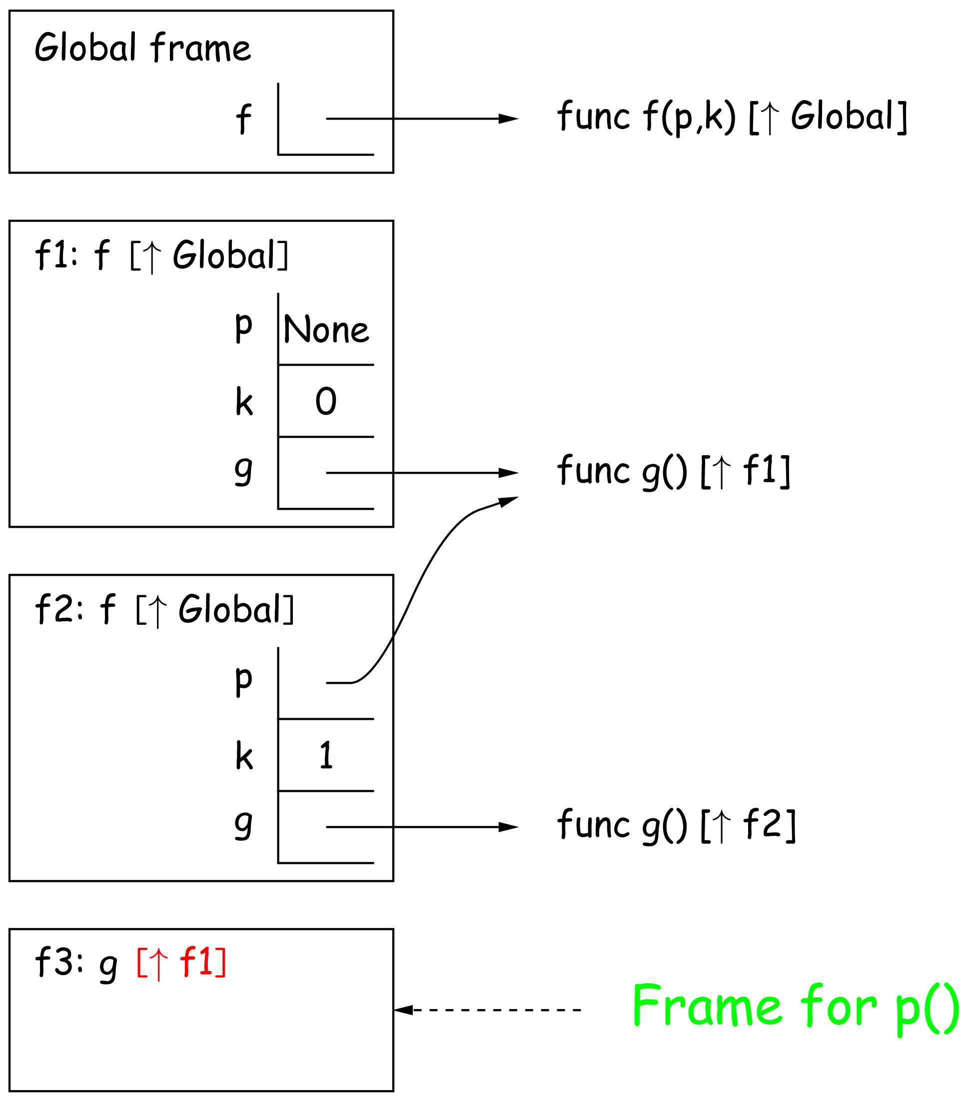
What is printed: (0, 1, or error) and why?

```
def f(p, k):
    def g():
        print(k)
    if k == 0:
        f(g, 1)
    else:
        p()
f(None, 0)
```

Answer (VI)

This prints 0. There are two local frames for f when p() is called (f1 and f2). The call to p() creates an instantiation of g whose parent is f1.

```
def f(p, k):
    def g():
        print(k)
    if k == 0:
        f(g, 1)
    else:
        p()
f(None, 0)
```



Decorators: Pythonic Use of Higher-Order Functions

- The syntax

```
@expr
def func(expr):
    body
```

is equivalent to ("syntactic sugar for")

```
def func(expr):
    body
func = (expr)(func)
```

- For example, our `ucb` module defines decorator `trace`. After

```
from ucb import trace
@trace
def mysum(x, y):
    return x + y
```

`mysum` will print its arguments and return value each time it is called.

- Usually, *expr* is a simple name, but it can be any expression that evaluates to a function that takes and returns a function.

Implement trace

```
def trace(func):
    """A decorator that accepts the same arguments
    and returns the same value as FUNC, but also
    prints the arguments and return value."""
    def afunc(arg):
        print("Call", func.__name__, "with", arg)
        v = func(arg)
        print(func.__name__, "returns", v)
        return v
    return afunc
```

Implement trace (Fancier Version)

- At the moment, `trace` handles only one-argument functions.
- To handle more general ones, we use two Python features:

```
def trace(func):
    """A decorator that accepts the same arguments
    and returns the same value as FUNC, but also
    prints the arguments and return value."""
    def afunc(*args):      # args is now a list of actual parameters
        print("Call", func.__name__, "with", args)
        v = func(*args)
        # Line above is like v = func(args[0], args[1], ...)
        print(func.__name__, "returns", v)
        return v
    return afunc
```

Design a Decorator

- I'd like a decorator that will check that the output of a function obeys some predicate:

```
@check_result(lambda x: x < 1000)
def compute(x):
    ...
    return whatever  # value of whatever must be < 1000.
```

- How would you define `check_result`?
- It must return a function that
  - Takes a function, say `func`, as input
  - Returns a function that takes the same arguments as `func` and returns the same value as `func` if that value satisfies `PRED`, but complains otherwise.

A Decorator That Checks Results

```
@check_result(lambda x: x < 1000)
def compute(x):
    ...
    return whatever  # value of whatever must be < 1000.
```

- We require that `check_result(lambda x: x < 1000)(compute)` returns a function that returns the same values as `compute`, but checks that they are less than 1000 first.
- Let's restrict ourselves to decorating 1-argument functions (like `compute`).
- The `check_result` function evidently takes a boolean function (predicate) as its argument:

```
def check_result(checker):
```

- And then returns *another* function that takes a function as its argument and returns a new one:

```
    def checked_func(func):
        ?
    return checked_func
```

Checking Decorator (continued)

- And this returned function must return *still another* function that calls the decorated function (such as `compute`) and then checks it:

```
def check_result(checker):
    def checked_func(func):
        def call_and_check(x):
            ?
        return call_and_check
    return checked_func
```

Checking Decorator (completed)

- Final result:

```
def check_result(checker):
    def checked_func(func):
        def call_and_check(x):
            result = func(x)
            if checker(result):
                return result
            else:
                raise ValueError("bad result") # indicate an error
        return call_and_check
    return checked_func
```

Higher-Order Functions at Work in Project #1

This project uses functions to represent aspects of playing a game:

- Strategy: Integer $\times$ Integer $\rightarrow$ Plan
  (your score, opponent score) $\mapsto$ how to play
- Dice: $\rightarrow$ Integer
  () $\mapsto$ random roll of die